#3

BUNDESREPUBLIK DEUTSCHLAND

Bescheinigung

Die Firma International Business Machines Corporation in Armonk, N.Y./V.St.A. hat eine Patentanmeldung unter der Bezeichnung

"Verfahren zur kompakten Darstellung von Informationspaketen und deren Speicherung oder Überrtragung"

am 17. Juli 1999 beim Deutschen Patent- und Markenamt eingereicht.

Die angehefteten Stücke sind eine richtige und genaue Wiedergabe der ursprünglichen Unterlagen dieser Patentanmeldung.

Die Anmeldung hat im Deutschen Patent- und Markenamt vorläufig das Symbol G 06 F 5/00 der Internationalen Patentklassifikation erhalten.

München, den 21. Februar 2000

Deutsches Patent- und Markenamt

Der Präsident

Im Auftrag

Aktenzeichen: 199 33 584.2

Waasmaier

B E S C H R E I B U N G

Verfahren zur kompakten Darstellung von Informationspaketen und deren Speicherung oder Übertragung

Gegenstand der Erfindung

Die vorliegende Erfindung beschreibt ein Verfahren zur kompakten Darstellung von Informationspaketen, insbesondere von Java-Objekten, zur Speicherung auf einem Speichermedium mit begrenztem Speicherplatz, insbesondere einer Chipkarte, oder zur Übertragung über ein Kanal mit begrenzter Bandbreite.

Stand der Technik

Speichern von Objekten ist ein wesentliches Thema bei der objektorientierten Software-Entwicklung. Ein Objekt stellt eine nach außen abgeschlossene Programmeinheit dar, die intern durch eine Reihe von lokalen Variablen in ihren Eigenschaften (Attributen) festgelegt ist. Der Inhalt dieser lokalen Variablen legt den Zustand eines Objekts außerhalb des Objekts sind jedoch weder die lokalen Variablen noch ihr Inhalt sichtbar. Der Zustand eines Objekts kann ausschließlich durch Ausführen eines seiner lokalen Unterprogramme verändert werden. Diese Unterprogramme werden auch Methoden genannt. Zur Ausführung einer Methode wird ein Objekt von einem anderen Objekt aufgefordert, indem das auffordernde Objekt eine entsprechende Nachricht schickt. Das angesprochene Objekt wird, sofern es die Botschaft versteht, die entsprechende Methode (lokales Unterprogramm) zur Ausführung bringen. Objektspeicherung beinhaltet das Ablegen von sowohl den Attributen, als auch von dem Zustand in denen sich Methoden/Algorithmen befinden, damit diese nach der Wiederherstellung des Objektes nahtlos fortgesetzt

werden können. Persistence, CORBA, ORB sind hierbei gängige Schlagworte.

Für Chipkarten existiert keine diesbezügliche Lösung. Üblicherweise werden Informationen auf Chipkarten in traditionellen Dateistrukturen abgelegt. Für die Anwendungsdaten der Host-Anwendung stehen auf der Chipkarte verschiedene einfache Dateitypen sowie Dateiverzeichnisse zur Verfügung. Daten zwischen Host und Karte werden über einzelne Befehlssequenzen (APDUs), denen die Daten in Form von Bytes angehängt werden, ausgetauscht. Diese dateiorientierten Karten haben heute die größte Verbreitung.

Eine gebräuchliche Methode von Java, Objektzustände zu speichern, ist die Objektserialisierung. Serialisierung beinhaltet, dass ein beliebiges Objekt in eine Sequenz von Bytes umgewandelt wird, die das Objekt und seinen momentanen Zustand repräsentiert. Dazu werden alle Attribute, deren Name sowie alle weiteren Objekte, auf die das Objekte referenziert, gespeichert. Mit diesen Informationen kann das Objekt unverändert wiederhergestellt werden. Die Wiederherstellung verläuft analog: Das Objekt, sowie alle anderen Objekte auf die das Objekt referenziert, wird instanziert und mit den von den Chipkarten gelesenen Daten initialisiert. Der Byte-Array, der das Objekt repräsentiert, kann sowohl auf JavaCards als auch auf gewöhnlichen dateiorientierten Karten problemlos gespeichert werden.

Nachteile Stand der Technik

Die bisher bekannten Verfahren zur Serialisierung und Speicherung von Objekten wurden für leistungsfähige Rechner konzipiert. Aufgrund ihres zu hohen Speicherbedarfs sind sie für das Ablegen von Objekten auf Chipkarten oder Speichermedien mit geringem Speicherplatz ungeeignet oder es können nur sehr kleine Objekte in serialisierter Form auf Chipkarten abgespeichert werden. Bei der Java Object

Serialization werden zum Beispiel selbst für kleine Objekte mit wenigen Attributen mehrere hundert Bytes benötigt.

Es ist daher Aufgabe der vorliegenden Erfindung, ein Verfahren bereitzustellen, um Informationspakete in einer einfachen und schnellen Weise in einer sehr kompakten Form auf einem Speichermedium mit begrenztem Speicherplatz ablegen zu können oder über einen Übertragungskanal mit geringer Bandbreite übertragen zu können, so dass das Informationspaket auf der Empfängerseite wiederhergestellt werden kann.

Diese Aufgabe wird durch die Merkmale in Anspruch 1 oder 2 gelöst. Weitere vorteilhafte Ausführungsformen sind in den Unteransprüchen niedergelegt.

Das erfinderische Verfahren ermöglicht die kompakte Darstellung komplexer Informationspakte (wie beispielsweise von Objekten im Sinne der OO Programmierung) ohne das die Informationspakete in einzelne Bestandteile (wie Attribute) zerlegt und einzeln optimiert werden muss.
Das Informationspaket wird als Ganzes (in einer "Black-Box") speicherplatzoptimiert abgebildet, so dass sie auf Speichermedien, wie zum Beispiel Chipkarten, abgelegt und zu einem beliebigen Zeitpunkt auf einem beliebigen System unverändert wiederhergestellt werden kann.
Die Wiedeherstellung erfolgt durch analoge, jedoch umgekehrte Anwendung des Algorithmus, wiederum in einer Black-Box.

Das Informationspaket braucht nicht anwendungsfallspezifisch und kontextspezifisch analysiert und in einzelne Informationen zerlegt werden.

Der Algorithmus wird auf das Informationspaket als Ganzes angewendet - ist anwendungsunabhängig (wenn die Stufe 3

angewendet wird, benötigt sie lediglich das zu verwendende Wörterbuch).
Die kompakte Darstellung enthält alle Informationen in sich, um an anderer Stelle unverändert wiederhergestellt zu werden.

Bei der Wiederherstellung des Informationspakets sind keine weitere Informationen - außer dem kompakten Informationspaket selbst - notwendig, um es wieder herzustellen (ggf. außer dem Wörterbuch, falls Stufe 3 angewendet wird).

Die vorliegende Erfindung wird anhand eines bevorzugten Ausführungsbeispiels im Zusammenhang mit FIG. 1 näher erläutert.

Informationspakete im Sinne der vorliegenden Erfindung sind beispielsweise:

1.) Objekte im Sinne der objektorientierten Programmierung (z.B. objektorientierte Programmiersprache Java, C++)

```
z.B. in Java Programm
public Object Ticket {
public String departure = "Stuttgart";
public String destination = "Frankfurt";
public Ticket(); /*Constructor*/
};
```

2.) Datenstrukturen (z.B. klassische Programmiersprache C)

```
z.B. in C-Programm
structure Ticket {
String departure = "Stuttgart";
String destination = "Frankfurt";
}
```

3.) Semantische Beschreibungen von Informationen (z.B. HTML, XML, Scripte)

z.B. in einer fiktiven(!) Script-Sprache:
<TICKET><STRING><DEPARTURE>"Stuttgart"<STRING><DESTINATION>"
Frankfurt"

Die Informationspakete bestehen aus Meta-Daten und Datenelementen. Meta-Daten sind diejenigen Daten, die zur Beschreibung der Datenelemente (z.B. Namen und Typ eines Datenelements) erforderlich sind. Datenelemente sind die eigentlichen Nutzdaten.

Speichermedien im Sinne der vorliegenden Erfindung sind beispielsweise:

- Festplatten, Chipkarten, Disketten, u.ä. Medien die dazu dienen den aktuellen Zustand der betrachteten Informationseinheit zwischenzuspeichern, um dieselbe Informationspakete zu einem anderen Zeitpunkt und in einem anderen System in unverändertem Zustand wiederherzustellen.

Das vorliegende erfinderische Verfahren zur kompakten Darstellung von Informationspaketen wird anhand der erfinderischen Verfahrensschritte 1-4 in Verbindung mit FIG. 1 dargestellt. Verfahrensschritte 3-4 stellen besondere Ausführungsformen der erfinderischen Methode dar, um den Komprimierungsfaktor zu erhöhen.

Das Serialisieren im Sinne der nachfolgenden Erfindung bedeutet, 1) dass die Meta-Daten und Datenelemente zu einer Dateneinheit (Informationspaket) zusammengeführt werden oder 2) dass nur die Datenelemente in einer Sequenz dargestellt werden, wobei der Empfänger die Zuordnung zwischen Meta-Daten und Datenelement durchführt. Im nachfolgenden wird vom Serialisierungsverfahren nach 1) ausgegangen.

1. Zusammenführen von Meta-Daten mit Datenelementen zu einem Informationspaket in einer Repräsentation, die alle für die

Wiederherstellung der Information notwendigen Angaben enthält. Zum Beispiel sind Meta-Daten Objektname, Objektversion, Objekttyp, Attribute, Attributtypen und aktuelle Attributewerte. Für das Java Objekt "Ticket" müssen gespeichert werden:

Objekt: Es ist ein Objekt vom Typ "java.lang.Object" mit dem Namen "Ticket" (ggf. mit einer Versionsangabe).
Das Objekt hat folgende Attribute:
Attribut1: Attribut "departure" vom Typ "java.lang.String" mit aktuellem Wert "Stuttgart"
Attribut2: Attribut "destination" vom Typ "java.lang.String" mit aktuellem Wert "Frankfurt"

2. In der Repräsentation der Informationspakete werden die darin verwendeten Kennungen (Namens-und Typkennungen), wie z.B. java.lang.String, java.lang.Object, etc., durch spezielle, definierte Identifikatoren oder Stellvertreter (z.B. tags gemäß TLV-Codierung nach ISO 8825 BER-TLV) dargestellt. Damit erhält man eine Darstellung, die beispielweise für den menschlicher Leser wie folgt aufbereitet dargestellt werden kann:

```
[(1,1,true) 321]
(
  [(15,1,false) 21] ( "tests.compress.Ticket" )
  [(7,2,true) 294]
  (
    [(18,1,false) 13] ( "departure" )
    [(8,2,false) 1] ( '01' )
    [(16,1,false) 9] ( "Stuttgart" )
    [(18,1,false) 15] ( "destination" )
    [(8,2,false) 1] ( '01' )
    [(16,1,false) 10] ( "Frankfurt" )
  )
)
```

Dabei bedeuten die Angaben folgendes:

[(tag-Identifikator,class,constructed flag)length](Inhalt)

tag-Identifikator dient zur Unterscheidung verschiedener tags, class dient zur Unterscheidung verschiedener Klassen von tags und constructed flag zeigt an, ob dem tag einfache Nutzdaten folgen oder wiederum weitere tags mit zugehörigen Nutzdaten.
Length umfaßt die Anzahl Bytes der folgenden Nutzdaten.
Inhalt bedeutet entweder zugehörige Nutzdaten oder weitere tags.

3. In einer besonderen Ausführungsform der vorliegenden Erfindung wird folgender zusätzlicher Verfahrensschritt angewandt:

Der vorhergehende Schritt 2 wird wiederholt, wobei jedoch nicht anwendungsunabhängige, sondern anwendungsabhängige Kennungen durch Identifikatoren (Stellvertreter) dargestellt werden. Vorzugsweise ist ein Wörterbuch notwendig, das für einen speziellen Anwendungskontext, spezielle Identifikatoren festlegt (z.B. für Reiseanwendungen könnte Tag xy für "departure" und Tag yz für "destination" stehen). Damit kann eine weitere Reduktion des für die Repräsentation des Informationspakets notwendigen Speichers erzielt werden. Damit erhält man eine Darstellung, die beispielweise wie folgt aussieht:

```
[(1,1,true) 321]
(
  [(15,1,false) 21] ( "tests.compress.Ticket" )
  [(7,2,true) 294]
  (
    [(26,1,false) 9] ( "Stuttgart" )
    [(26,1,false) 10] ( "Frankfurt" )
  )
```

)

4. In einer weiteren besonderen Ausführungsform wird folgender zusätzliche Verfahrensschritt angewandt:

Die erzielte Darstellung wird mit Hilfe gängiger Komprimierungsalgorithmen, z.B. ZLIB, noch einmal komprimiert.

Insgesamt werden durch die Schritte 1-4 eine kompakte Repräsentation der Informationspakete erzielt, die ca. 30% weniger Speicherplatz in Anspruch nimmt, als eine gewöhnliche Serialisierung mit bestmöglicher Komprimierung durch Komprimierungsalgorithmen wie ZLIB.

Trotz der Komprimierung des Informationspakets sind alle Informationen enthalten, um das Paket an anderer Stelle zu einem anderen Zeitpunkt unverändert wiederherstellen zu können.

Das vorliegende erfinderische Verfahren wird anhand von Java-Objekten dargestellt.

Anwendungsbeispiel:

Im nachfolgenden Ausführungsbeispiel wird die Java-Object-Serialisierung durch ein Serialisierungsverfahren mit TLV (Tag Length Value) Strukturen ersetzt. Bei der Speicherung von Objekten auf einer Chipkarte mit Hilfe von TLV Strukturen wird durch die Vorverarbeitung der „tags" Speicherplatz eingespart.

Die Einsparung von Speicherplatz erfolgt in vier Schritten:

1. Erzeugung der Objektdarstellung (Zusammenführen von Meta-Daten und Datenelemente, Bilden einer Sequenz)

2. Codierung der anwendungsunabhängigen Kennungen in tags, z.B. für Objekte, Attribute, Namen, Klassen,...

3. Codierung von anwendungsabhängigen Kennungen in tags

4. Zusätzliche Komprimierung

Vorgehensweise:

Ein Objekt wird auf der Chipkarte wie folgt dargestellt:

- Name des Objekts
- Typ des Objekts (string, integer, object)
- Version
- Attribute des Objekts (Attribute und Werte der Attribute können selbst wieder Objekte sein).

Beispiel:

In folgendem Beispiel können alle groß geschriebenen Wörter durch „tags" dargestellt werden. Zu jedem „tag" gehört ein Wert und die Länge dieses Werts.

```
NAME("BC")
TYPE(OBJECT)
OBJECT (
  NAME("com.ibm.businesscard.BusinessCard")
  TYP (object)
  VERSION("1.1")
  ATTRIBUTE(
     NAME("Name")
     TYPE(STRING)
     VALUE("Mustermann")
     )
  ATTRIBUTE(
     NAME ("Address")
```

```
TYPE(OBJECT)
      OBJECT(
      NAME("com.ibm.business.Address")
      VERSION("1.0")
      ATTRIBUTE(NAME("Street") TYPE(STRING)
      VALUE("Schoenaicher Str. 220"))
      ATTRIBUTE (NAME("City") TYPE(STRING)
      VALUE("Boeblingen"))
   )

  )
```

In einer Tabelle, die Bestandteil des erfinderischen Komprimierungsalgorithmus ist, ist die Zuordnung zwischen „Identifikator" (Stellvertreter) und einem „tag" abgelegt. Die „tags" werden vorzugsweise gemäß BER-TLV bestimmt. Als „tags" werden vorzugsweise diejenigen Identifikatoren definiert, die anwendungsunabhängig sind.
In der nachfolgenden Tabelle ist eine beispielhafte Zuordnung abgebildet.

Tags	Klasse	ID	Constructed	Encoding(hex)
OBJECT	2	0	true	A0
CLASS	2	1	false	81
VERSION	2	2	false	82
ATTRIBUTE	2	3	true	A3
NAME	2	4	false	84
TYPE	2	5	false	85
VALUE	2	6	false	86

Typ	ID
INTEGER	0
STRING	1
OBJECT	2
...	

Für das Beispiel:

A0(tag-hex für object)
xx

84(tag-hex für name) 32(Länge in Byte)"
com.ibm. businesscard.BusinessCard"
82 (tag-hex für version) 03(Länge in Byte)
"1.0"
A3(tag-hex für Attribute)35(Länge des
Attributs in Byte)Wert:84,85,86
84(tag-hex für name)04(Länge des name) „Name"
(Wert)
85(tag-hex für Typ) 01 (Länge des Typs)
Wert:1
86 (tag -hex für Value) 10(Länge des
Values)Wert:"Mustermann"
A3(tag-hex für Attribute)35(Länge des
Attributs in Byte)Wert:84,85,86
84(tag-hex für name)04(Länge des name) Wert
„Address„
85(tag-hex für Typ) 01 (Länge des Typs)
Wert:2

.
.
.

Um die Komprimierung noch zusätzlich zu erhöhen, kann folgende zweite Komprimierungsphase angewendet werden:

Das nach dem oben beschriebenen Verfahren erzielte Ergebnis wird anschließend nach vordefinierten, anwendungsabhängigen Namens-und Typkennungen für Objekte durchsucht. Für diesen Verfahrensschritt stehen spezielle „tags" zur Verfügung, mit denen diese Kennungen platzsparend dargestellt werden können. Dadurch wird eine weitere Komprimierung erreicht.

Die Zuordnung zwischen diesen anwendungsabhängigen Kennungen und den „tags", durch den diese dargestellt werden können, muß natürlich allen Beteiligten, die Informationen lesen oder schreiben wollen, bekannt sein. Denkbar ist es, verschiedene Zuordnungstabellen für verschiedene Anwendungskontexte zu definieren. Es werden je nach Kontext öfter benötigte Strukturen und Typen festgelegt, um redundante Informationsspeicherung zu vermeiden. Beispielsweise könnte ein Anwendungskontext „Travel", tags wie Sitzplatz, Zielort, Abfahrtszeit, etc. definieren.

Aufbau, Definition und Länge von „tags" sind in den Normen ISO/IEC 7816, bzw. 8825 festgelegt. Für diese neue Anwendungsmöglichkeit von TLV Strukturen ist es evtl. erforderlich, die maximale Länge der „tags" zu erhöhen. Das folgende Beispiel soll diese zweite Komprimierungsstufe erläutern:

Der Name „com.ibm.businesscard.BusinessCard" ist in dem Anwendungskontext definiert und kann wie folgt verkürzt dargestellt werden:

A0(tag-hex für object)
xx
84(tag-hex für name) 32(Länge in Byte) BF
82 (tag-hex für version) 03(Länge in Byte) "1.0"
A3(tag-hex für Attribute)35(Länge des Attributs in Byte)Wert:84,85,86
84(tag-hex für name)04(Länge des name) „Name" (Wert)
85(tag-hex für Typ) 01 (Länge des Typs) Wert:1
86 (tag -hex für Value) 10(Länge des Values)Wert:"Mustermann"
A3(tag-hex für Attribute)35(Länge des Attributs inByte)Wert:84,85,86
84(tag-hex für name)04(Länge des name) Wert „Address„
85(tag-hex für Typ) 01 (Länge des Typs) Wert:2
.

Tag	Klasse	ID	Constructed	Type	Class
BF	2	100	true	Name	
com.ibm...BusinessCard					

Die Zuordnung zwischen tag und Identifikator muß dem Empfänger bekannt sein oder muß mitübertragen werden (z.B. die Tupel der Form <Tag,Klasse,ID,Constructed,Type,Class>)

Evaluierung des Methode

Zur Abschätzung des Potentials des beschriebenen erfinderischen Verfahrens wurden verschiedene Objekte entsprechend des erfinderischen Verfahrens komprimiert und die dabei erreichte Komprimierung bewertet. Hierbei wurden insbesondere Ergebnisse, die mit Hilfe anderer, bekannter Komprimierungsalgorithmen erreicht wurden, zum Vergleich herangezogen.

Beispiel 1:

Nachfolgendes Objekt beschreibt ein Flugticket:

```
public class Ticket extends Object implements Serializable
{
public String passengerName ="Hans Mustermann";
public String passengerAddress = "Schoenaicher Strasse 220";
public String  passengerCity = "71032 Boeblingen";
public String  departureCity= "Stuttgart";
public String  destinationCity= "Whitehorse";
public Integer flightNumber= new Integer(478);
public String  airline= "LH";
public String  date= "24.12.99";
```

```
public String  departureTime= "11:00";
public Integer ticketPrice= new Integer(1200);
public Integer status  = new Integer(1) ;

  // Default Constructor
  public Ticket () {
  }
}
```

Beispiel 2:

Dieses Flugticket enthält die Passagierinformationen in einem gesonderten Objekt. Die Methodik muß also auf verschachtelte Objekte angewendet werden, was eine Rekursion erfordert.

```
public class Ticket2 extends Object implements Serializable
{
  public Passenger  passenger
          = new Passenger ("Hans Mustermann",
          "Schoenaicher Strasse 220",
          "71032 Boeblingen");
  public String  departureCity= "Stuttgart";
  public String  destinationCity= "Whitehorse";
  public Integer flightNumber= new Integer(478);
  public String  airline= "LH";
  public String  date= "24.12.99";
  public String  departureTime= "11:00";
  public Integer ticketPrice= new Integer(1200);
  public Integer status  = new Integer(1) ;

  // Default Constructor
  public Ticket2 () {
  }
}

public class Passenger extends Object implements Serializable
{
```

```
  public String  passengerName ="Hans Mustermann";
  public String  passengerAddress ="Schoenaicher Strasse 220";
  public String  passengerCity ="71032 Boeblingen";

  // Default Constructor
  public Passenger () {
  }
}
```

Bewertung der erreichten Komprimierung:

Die nachfolgende Tabelle zeigt den für die Darstellung der Beispielsobjekte benötigten Speicherplatz bei konventioneller Serialisierung und bei Anwendung der ersten Stufe der hier vorgestellten kompakten Darstellung. Es wurde jeweils durch Anwendung des "ZLib Data Compression Algorithm" mit der Einstellung "Best Compression" noch eine zusätzliche Verbesserung des Ergebnisses erreicht.

Die Anwendung der zweiten Stufe der hier vorgestellten Methode würde eine weitere Verbesserung erzielen, wurde jedoch nicht in diese Evaluierung einbezogen.

	Konventionelle Objektserialisierung in Java	Konventionelle Objektserialisierung in Java mit zusätzcher Komprimierung (ZLib Data Compr. Algorithm)	Kompakte Darstellung der Objekte (1.Stufe)	Kompakte Darstellung der Objekte (1.Stufe) mit zusätzlicher Komprimierung (ZLib Data Compr. Algorithm)
Beispiel 1	642 Byte	353Byte	325 Byte	245
Beispiel 2	725 Byte	392Byte	374 Byte	268

In Beispiel 1 wird eine Verdichtung um 31 % gegenüber dem mit ZLib komprimierten serialisiertem Objekt erzielt. In Beispiel 2 beträgt die Verdichtung 32 %. Verschiedene

weitere untersuchte Objekte ergaben Werte zwischen 16 % und 40%.

Die Methode zielt im Wesentlichen darauf ab, die Meta-Daten einer Objektrepräsentation zu minimieren. Das bedeutet, daß der Komprimierungsfaktor steigt, je kürzer die Datenlängen der einzelnen Attribute eines Objektes sind. Bei sehr großen Dateninhalten je Attribut, ist der Nutzen etwas geringer.

Vorteile der Lösung

Durch das Ersetzen der Java Objektserialisierung durch das beschriebene Verfahren können Objekte kompakt und platzsparend auf Chipkarten abgespeichert werden. Dies vereinfacht zum einen, den Programmieraufwand, um Objektdaten auf einer Chipkarte zu speichern und auszulesen, zum anderen erhält man durch Anwendung dieses Verfahrens Objektrepräsentationen, die sehr klein sind. Es wurde eine Reduzierung des Speicherplatzes um 25- 50 % gegenüber der klassichen Objektserialisierung festgestellt.
Die zweite Stufe des vorgestellten Verfahrens erzielt eine zusätzliche Komprimierung. Sie setzt jedoch eine branchenspezifische, einheitliche Definition von Anwendungsattributen voraus.

So könnten Anwendungen der Tourismusbranche, für ihren Kontext eine Reihe von Attributen wie Reisepreis, Abflugdatum, Passagiername, etc. durch standardisierte „tags" ersetzen.

Ein weiterer Vorteil dieser Lösung ist der Brückenschlag zwischen der Methodik der Objektserialisierung und der Benutzung des TLV Verfahrens.

Im nachfolgenden wird eine konkrete Implementierung der vorliegenden Erfindung dargestellt.

```
/*************************************************************
*     serialize
*     Serializing the state of a given object, using common Java
      methods
*************************************************************/
  public byte[] serialize(Object object)
  {
    byte[] result = null;

    try {

      // create a byte array and a output stream
     ByteArrayOutputStream buffer = new
     ByteArrayOutputStream();
     //FileOutputStream buffer = new FileOutputStream("test");
      ObjectOutputStream out=new ObjectOutputStream(buffer);

      // write to stream
      out.writeObject(object);
      out.close();

      // result
      result = buffer.toByteArray();

    } catch (Exception e) {
      System.out.println("Shit happens");
    }

    return result;

  }
```

```
/*************************************************************
* restore
* Restoring a serialized Jav Object, using common Java-methods
*************************************************************/
  public Object restore(byte[] data){

    Object object = null;

    try {
      // create a byte array input stream
     ByteArrayInputStream buffer = new
     ByteArrayInputStream(data);
     ObjectInputStream in = new ObjectInputStream(buffer);

      // read from stream
      object = in.readObject();
      in.close();

    } catch (Exception ex) {
      System.out.println("Shit happens");
    }

    return object;
  }
 /*************************************************************
  *  simpleZip
  *  Compressing bytes using the common ZLIB Algorithmus with
     option BEST_COMPRESSION
 *************************************************************/
  public byte[] simpleZip(byte[] data) {

    byte[] result = null;

   Deflater deflater = new Deflater(Deflater.BEST_COMPRESSION,
   true);
   deflater.setInput(data,0,data.length);
```

```
    deflater.finish();

    byte[] buffer = new byte[5000];
    int length  = deflater.deflate(buffer);
    result = new byte[length];
    System.arraycopy(buffer, 0, result, 0, length);

    return result;
  }

/************************************************************
  * simpleUnZip
  * Unzipping compressed bytes using the ZLIB Algorithmus
 ************************************************************/
  public byte[] simpleUnZip(byte[] data) {

    byte[] result = null;

    try {
      Inflater inflater = new Inflater(true);
      inflater.setInput(data,0,data.length);

      byte[] buffer = new byte[5000];
      int length  = inflater.inflate(buffer);
      result = new byte[length];
      System.arraycopy(buffer, 0, result, 0, length);
    } catch (Exception ex) {
      System.out.println("Shit happens");
    }

    return result;
  }
```

```
/*************************************************************
*    specialSerialize
*    Serializing a object using TLV structure to replace
     redundant information
*************************************************************/
  public TLV specialSerialize(Object object) {

    TLV result = null;

    System.out.println("\nspecial serialize ... \n");

    // just a simplified implementation without nested classes!
    try {

      // 1.) create the attributeTLV Containing the attributes
      TLV attributeTLV = null;
      Field[] fields=object.getClass().getFields();

      for (int i=0; i<fields.length; i++)
      {
     // get field name
     String fieldName = fields[i].getName();
     System.out.println("optimized serialization of field
     "+fieldName);
     TLV nameTLV = new TLV ( TAG_NAME,fieldName.getBytes()) ;
     if (attributeTLV==null)
     attributeTLV = new TLV (TAG_ATTRIBUTE, nameTLV);
     else
     attributeTLV.add(nameTLV);

     // get fieldValue and Type
     Object fieldValue = fields[i].get(object);
     Class  fieldType = fields[i].getType();
     TLV typeTLV=null;
     TLV valueTLV = null;
```

```
        // Strings ...
        if
    (fieldType.isAssignableFrom(Class.forName("java.lang.St
    ring"))) {
    typeTLV = new TLV ( TAG_TYPE , TYPE_STRING) ;
    valueTLV = new TLV ( TAG_VALUE ,
    ((String)fieldValue).getBytes()) ;
    }
    else
    {
    // Integers ...
    if   (fieldType.isAssignableFrom(Class.forName("java.la
         ng.Integer"))) {
         typeTLV = new TLV ( TAG_TYPE , TYPE_INTEGER) ;
valueTLV = new TLV ( TAG_VALUE ,      ((Integer)fieldValue
                                      ).intValue()) ;
         } else {
           // other Objects ...
           typeTLV = new TLV ( TAG_TYPE , TYPE_OBJECT) ;
          // o.k. go into recursion ...
    TLV embeddedObjectTLV = specialSerialize(fieldValue) ;
    valueTLV = new TLV ( TAG_VALUE,
    embeddedObjectTLV.toBinary());
         }
       }

       attributeTLV.add(typeTLV);
       attributeTLV.add(valueTLV);
     }

     // 2.) create the Class TLV
    TLV classTLV = new TLV (TAG_CLASS,
    object.getClass().getName().getBytes());
    System.out.println("optimized serialization of class
    "+object.getClass().getName());
```

```
        // 3.) create the wrapper around the attributes
        TLV objectTLV = new TLV (TAG_OBJECT, classTLV);
        // ****TO DO****TO DO objectTLV.add(versionTLV);
        objectTLV.add(attributeTLV);


       // we've got it now: objectTLV is our specialZipped
       Object!
       result = objectTLV;


      } catch (Exception ex) {
        System.out.println("Shit happens");
        ex.printStackTrace();


      }

      return result;
    }

/*************************************************************
* specialRestore
* Restoring an object serialized with the method
specialSerialize()
*************************************************************/
    public Object specialRestore(TLV data) {

      Object result = null;

      System.out.println("\nspecial restore ... \n");

      try
      {
        TLV objectTLV = data;
```

```
 // unzip the object itself
 TLV classTLV = objectTLV.findTag(TAG_CLASS, null);
 Class objectClass = Class.forName(new  StringclassTLV.
                                        valueAsByteArra
                                        y()));
System.out.println("restoring Class
"+objectClass.getName());
 result = objectClass.newInstance();

 // restore the version ...
 // ****TO DO****

 // restore all values of the object
 TLV attributeTLV = objectTLV.findTag(TAG_ATTRIBUTE,
null);

 // loop through all  attributes of the attribute TLV
 TLV lastProccessedNameTLV = null;
 TLV lastProccessedTypeTLV = null;
 TLV lastProccessedValueTLV = null;
 TLV aNameTLV = null;
 TLV aTypeTLV = null;
 TLV aValueTLV = null;
 do
 {
   // find the next subTLV in the attributeTLV
   aNameTLV =  attributeTLV.findTag(TAG_NAME,lastProcce
               ssedNameTLV);
aTypeTLV = attributeTLV.findTag(TAG_TYPE,
lastProccessedTypeTLV);
aValueTLV = attributeTLV.findTag(TAG_VALUE,
lastProccessedValueTLV);

if ((aNameTLV!=null)&&(aTypeTLV!=null)&&(aValueTLV!=null))

{
// restore the field "aName" with the value "aValue"
```

```
        String aName = new String (aNameTLV.valueAsByteArray());
            System.out.println("restoring Field "+ aName);
            Field field = objectClass.getField(aName);
            int aType = aTypeTLV.valueAsNumber();

            if (aType == TYPE_STRING) {
              // STRING...
        String aValue = new String
        (aValueTLV.valueAsByteArray());
        field.set(result, aValue);
        } else {
                  if (aType == TYPE_INTEGER) {
                  // INTEGER ...
                   Integer aValue = new Integer
                  (aValueTLV.valueAsNumber());
                   field.set(result, aValue);
              } else {
                // OBJECT ...
                // o.k. go into recursion ...
                Object aValue = specialRestore(new TLV
(aValueTLV.valueAsByteArray()));
                     field.set(result, aValue);
                  } // else integer
              } // else string
            } // if TLV's !=null
            lastProccessedNameTLV = aNameTLV;
            lastProccessedTypeTLV = aTypeTLV;
            lastProccessedValueTLV = aValueTLV;

        } while ((aNameTLV!=null));

      } catch (Exception ex) {
        System.out.println("Shit happens");
        ex.printStackTrace();
      }
      return result;
    }
```

P A T E N T A N S P R Ü C H E

1. Verfahren zum Erzeugen einer kompakten Darstellung eines Informationspakets, wobei das Informationspaket zumindest aus Meta-Daten und dazugehörige Datenelemente und/oder Meta-Daten und dazugehörige Informationspakete besteht, wobei die Meta-Daten zumindest aus Namens- und Typkennungen des Datenelements oder Namens- und Typkennungen des Informationspakets bestehen, gekennzeichnet durch folgende Schritte:

   a) Anordnen der Informationspakete in einer Sequenz

   b) Durchsuchen der Meta-Daten nach definierten anwendungsunabhängigen Namens- und Typenkennungen (Kennungen)

   c) Darstellen der nach Schritt b) gefundenen Kennungen durch definierte Stellvertreter mit geringerem Speicherbedarf.

2. Verfahren zum Erzeugen einer kompakten Darstellung einer Struktur von Meta-Daten und Datenelementen, wobei die Zuordnung von Meta-Daten zu Datenelementen oder Meta-Daten zu einer Unterstruktur einer Struktur durch ein Programm erfolgt, wobei die Meta-Daten zumindest aus Namens- und Typkennungen (Kennungen) des Datenelements oder Kennung der Unterstruktur der Struktur bestehen, gekennzeichnet durch folgende Schritte:

   a) Zusammenführen von Meta-Daten und zugehörigen Datenelementen oder Meta-Daten mit der zugehörigen Unterstruktur zu einem Informationspaket

b) Anordnen der Informationspakete in einer Sequenz

c) Durchsuchen der Meta-Daten nach definierten anwendungsunabhängigen Kennungen

d) Darstellen der nach Schritt c) gefundenen Kennungen durch definierte Stellvertreter mit geringerem Speicherbedarf.

3. Verfahren nach Anspruch 1 oder 2 gekennzeichnet durch folgenden weiteren Schritt:

e) Speichern des Ergebnisses nach Schritt a-d auf einem Speichermedium

4. Verfahren nach Anspruch 1 oder 2 gekennzeichnet durch folgenden weiteren Schritt:

e) Übertragen des Ergebnisses nach Schritt a-d auf ein Datenverarbeitungsgerät.

5. Verfahren nach Anspruch 1 dadurch gekennzeichnet, dass die Meta-Daten nach Anspruch 3 oder 4 nicht mit übertragen werden, sondern die Zuordnung bei der Wiederherstellung durch ein Programm erfolgt.

6. Verfahren nach Anspruch 1 dadurch gekennzeichnet, dass das Informationspaket ein Objekt ist, das zumindest folgende Datenelemente mit folgenden anwendungsunabhängigen Kennungen enthält:

Objektname, Objekttyp und Objekt-Attribute.

7. Verfahren nach Anspruch 1 dadurch gekennzeichnet, dass das Informationspaket ein Objekt ist, das folgende Datenelemente mit folgenden anwendungsunabhängigen Kennungen enthält:

Objektname, Objekttyp, Objektversion und Objekt-Attribute.

8. Verfahren nach Anspruch 4 dadurch gekennzeichnet, dass das Informationspaket ein Java-Objekt ist.

9. Verfahren nach Anspruch 1 dadurch gekennzeichnet, dass das Informationspaket ein XML (Extendable Markup Language) ist.

10. Verfahren nach Anspruch 1 oder 2 dadurch gekennzeichnet, dass die anwendungsunabhängigen Kennungen Objektname, Objekttyp, Objektversion und Objekt-Attribute durch definierte Stellvertreter in einer TLV-Codierung dargestellt werden.

11. Verfahren nach Anspruch 1 oder 2 dadurch gekennzeichnet, dass die anwendungsunabhängigen Kennungen Objektname, Objekttyp, Objektversion und Objekt-Attribute durch definierte Stellvertreter in der TLV-Codierung nach ISO 8825 Basic Encoding Rules dargestellt werden.

12. Verfahren nach Anspruch 1 oder 2 dadurch gekennzeichnet, daß das Informationspakt eine Datenstruktur ist, die Datenelemente mit folgenden anwendungsunabhängigen Kennungen enthält:

Name, Typ, Version und Attribute des Datenelements

13. Verfahren nach Anspruch 1 oder 2 dadurch gekennzeichnet, daß die Schritte a)-d) durch ein Programm durchgeführt werden, wobei im Programm eine Tabelle zur Zuordnung von anwendungsunabhängigen Kennungen mit den zugeordneten Stellvertretern enthalten ist.

14. Verfahren nach Anspruch 1 oder 2 gekennzeichnet durch folgende weiteren Schritte:

    aa) Durchsuchen der Meta-Daten nach definierten anwendungsabhängigen Kennungen

    bb) Darstellen der nach Schritt aa) gefundenen anwendungsabhängigen Kennungen durch definierte Stellvertreter mit geringerem Speicherbedarf

    cc) Speichern des Ergebnisses nach Schritt aa)-bb) auf einem Speichermedium oder Übertragen des Ergebnisses nach Schritt aa)-bb) auf ein Datenverarbeitungsgerät.

15. Verfahren nach Anspruch 14 dadurch gekennzeichnet, daß für jede Anwendung eine eigene Tabelle mit definierten anwendungsabhängigen Kennungen und zugeordneten Stellvertreter geladen wird anhand der Verfahrenschritt bb) nach Anspruch 14 erfolgt.

16. Verfahren nach Anspruch 1 oder 2 dadurch gekennzeichnet, daß die Stellvertreter in Aufbau, Definition und Länge durch die Normen ISO/IEC 7816 oder 8825 festgelegt werden.

17. Verfahren nach Anspruch 16 dadurch gekennzeichnet, dass der Stellvertreter maximal 2 Byte Speicherplatz einnimmt.

18. Verfahren nach Anspruch 16 dadurch gekennzeichnet, dass sich der Stellvertreter aus Klasse, constructed flag und ID zusammensetzt.

19. Verfahren nach Anspruch 14 gekennzeichnet durch folgenden weiteren Schritt:

aaa) Anwenden eines gängigen Komprimierungsalgorithmus auf das Ergebnis nach Schritt aa)-bb) des Anspruchs 14

bbb) Speichern des Ergebnisses nach Schritt aaa) auf einem Speichermedium oder Übertragen des Ergebnisses nach Schritt aaa) auf ein Datenverarbeitungsgerät.

20. Verfahren nach Anspruch 19 dadurch gekennzeichnet, dass der Komprimierungsalgorithmus der ZLIP-Komprimierungsalgorithmus ist.

21. Verfahren nach Anspruch 1 oder 2 dadurch gekennzeichnet, dass das Speichermedium eine Chipkarte oder ein mobiler Datenträger ist oder dass der Übertragungskanal eine geringe Bandbreite hat.

22. Chipkarte enthaltend zumindest einen nichtflüchtigen Speicher, dadurch gekennzeichnet, dass das Ergebnis des Verfahrens nach Anspruch 1 bis 21 im nichtflüchtigen Speicher in einer Datei abgespeichert ist.

23. Vorrichtung enthaltend zumindest:
a) Datenverarbeitungsgerät
b) Kommunikationsmittel
c) Chipkarte, wobei über das Kommunikationsmittel Daten zwischen Datenverarbeitungsgerät und Chipkarte austauschbar sind, dadurch gekennzeichnet, dass auf dem Datenverarbeitungsgerät ein Programm zur Steuerung eines Verfahrens nach Anspruch 1 bis 21 installierbar ist und das Ergebnis des Verfahrens nach Anspruch 1 bis 21 auf der Chipkarte abgespeichert ist.

24. Computerprogrammprodukt, das im internen Speicher eines digitalen Rechner gespeichert werden kann, enthaltend Teile von Softwarecode zur Ausführung des Verfahrens nach Anspruch 1-21, wenn das Produkt auf dem Rechner ausgeführt wird.

Z U S A M M E N F A S S U N G

Das erfinderische Verfahren basiert darauf, Informationspakte-insbesondere Java-Objekte- vor ihrer Übertragung bzw. Speicherung auf einem Speichermedium, in serialisierter Form darzustellen, auf anwendungsunabhängige Kennungen zu untersuchen und die anwendungsunabhängigen Kennungen durch Stellvertreter mit geringerem Speicherbedarf darzustellen. In einer weiteren Ausführungsform werden auch die anwendungsabhängigen Kennungen durch spezielle Stellvertreter dargestellt. Das Ergebnis dieses Verfahrens ermöglicht die kompakte Darstellung komplexer Informationspakte, ohne dass die Informationspakete in einzelne Bestandteile zerlegt und einzeln optimiert werden müssen.

Das Informationspaket wird als Ganzes speicherplatzoptimiert abgebildet, so dass sie auf Speichermedien, wie zum Beispiel Chipkarten, abgelegt und zu einem beliebigen Zeitpunkt auf einem beliebigen System unverändert wiederhergestellt werden kann.

Die Wiedeherstellung erfolgt durch analoge, jedoch umgekehrte Anwendung des Algorithmus, wiederum in einer Black-Box.

Das Informationspaket braucht nicht anwendungsfallspezifisch und kontextspezifisch analysiert und in einzelne Informationen zerlegt werden. Durch die Anwendung dieses Verfahrens kommt es zu einer ca. 30% Verdichtung der Darstellung der Daten eines Informationspakets.

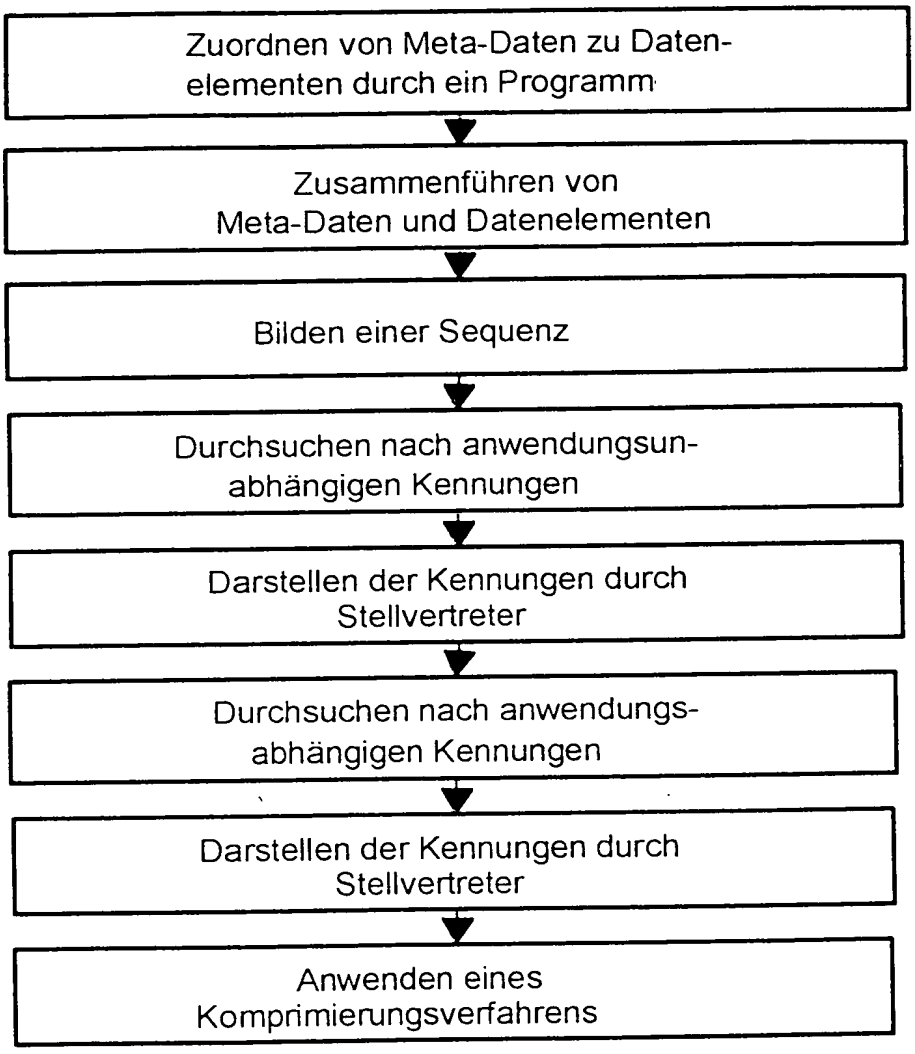
Zuordnen von Meta-Daten zu Daten-elementen durch ein Programm
Zusammenführen von Meta-Daten und Datenelementen
Bilden einer Sequenz
Durchsuchen nach anwendungsun-abhängigen Kennungen
Darstellen der Kennungen durch Stellvertreter
Durchsuchen nach anwendungs-abhängigen Kennungen
Darstellen der Kennungen durch Stellvertreter
Anwenden eines Komprimierungsverfahrens

FIG. 1